Dynamic Maintenance of Delaunay Triangulations

Thomas Kao*
David M. Mount†
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland

Alan Saalfeld
Bureau of the Census

Abstract

We describe and analyze the complexity of a procedure for computing and updating a Delaunay triangulation of a set of points in the plane subject to incremental insertions and deletions. Our method is based on a recent algorithm of Guibas, Knuth, and Sharir for constructing Delaunay triangulations by incremental point insertion only. Our implementation features several methods that are not usually present in standard GIS algorithms. Our algorithm involves:

Incremental update: During point insertion or deletion only the portion of the triangulation affected by the insertion or deletion is modified.

Randomized methods: For triangulation building or updates involving large collections of point, randomized techniques are employed to improve the expected performance of the algorithm, irrespective of the distribution of points.

Persistence: Earlier versions of the triangulation can be recovered efficiently.

1 Introduction

The Voronoi diagram and its dual, the Delaunay triangulation, are among the most useful structures that can be derived from a finite set of n points in the plane. These structures have long been recognized as being very useful in automated cartographic applications [6, 8]. Although it is known that these structures can be computed in worst case $O(n \log n)$ time [2, 5], it is widely felt that the implementation of these algorithms involves a significant amount of programming effort. As a consequence many implementors have settled for a simple incremental algorithm, which builds the

*The work of this author was partially supported by the Bureau of the Census under grant JSA 89–32.

†The work of this author was partially supported by the Bureau of the Census under grant JSA 89–32 and by the NSF under grant CCR–8908901.

diagram site by site [3, 5]. Although there are instances in which this algorithm runs in $O(n^2)$ worst case time, it is often observed that the performance of the incremental algorithm is rarely as bad as this quadratic bound suggests.

Recently Guibas, Knuth, and Sharir have given a theoretical explanation of this phenomenon [4]. They analyzed the complexity of the simple incremental algorithm for Delaunay triangulations combined with an novel technique for locating the triangle of the triangulation which contains a given point. They showed that, irrespective of the distribution of points, this algorithm operates in $O(n \log n)$ expected time provided that the points are inserted in random order. (Here the expectation is over the possible insertion orders.) We extend their result by giving an algorithm which can incrementally maintain a Delaunay triangulation through a sequence of insertions as well as deletions.

An incremental algorithm is said to have an *amortized time complexity* of $f(n)$ if the total cost of any sequence of N operations divided by N is $O(f(n))$, even though a single operation may have cost much greater than $f(n)$. We show that, given a base set of points, any sequence of insertions and deletions to the Delaunay triangulation can be performed on-line in expected amortized time $O(\log n)$ per insertion or deletion under the assumptions that (1) for insertion, each of the base points not present in triangulation is equally likely to be inserted, and (2) for deletion, each of the points present in the triangulation is equally likely to be deleted. Here n reflects the number of points present in the triangulation at the time of the update. No assumptions are made about the distribution of the base points.

Our algorithm has an interesting type of *persistence* property. In particular, we are able to reconstruct any earlier version of the triangulation more efficiently than the naive method of simply reversing the recent history of insertions and deletions.

We have implemented our algorithms in order to establish the actual efficiency, which was established theoretically by Guibas, Knuth and Sharir. We present a number of observations on the algorithm and its practical performance, and in particular we consider how the algorithm performs when the assumption of random insertion and deletion is violated.

The remainder of the paper is organized as follows. In Section 2 we describe the incremental insertion of Guibas, Knuth, and Sharir (for the sake of completeness). In Section 3 we describe the deletion algorithm and analyze its expected case complexity and in Section 4 we consider the complexity of sequences of insertions and deletions and how to keep the search structure balanced through such a sequence. In Section 5 we discuss our implementation of the algorithm and provide a number of graphs displaying the essential elements of the algorithm which determine its complexity.

2 Incremental Insertion

In this section we review the basic incremental algorithm as presented by Guibas, Knuth, and Sharir [4]. The algorithm is quite simple. Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of points and let $D(P)$ denote the Delaunay triangulation of this point set. For points $a, b, c \in P$, let $\triangle abc$ denote the triangle (not necessarily in the triangulation) determined by these points. We consistently label the vertices of triangles in counterclockwise order. For simplicity, we make the usual general position assumptions that no three points are colinear and that no four points are cocircular. These assumptions are handled in our implementation, but we omit discussion of them here since they clutter the presentation with undue detail.

We assume that the point coordinates have been normalized so that they lie within the interior of the unit square. It is assumed that the four corners of the unit square are always elements of P, and these four points are not eligible for deletion. When the algorithm is initiated, the Delaunay triangulation consists of two triangles formed by adding a diagonal through the unit square. (These four points actually violate our general position assumption, implying that either of the two diagonals could be used.)

The insertion procedure operates as follows. Suppose that p is a new point to be added to the triangulation. By a point location method (to be described later) determine the triangle $\triangle abc$ of $D(P)$ which contains this point. Replace the triangle $\triangle abc$ with three triangles $\triangle pab$, $\triangle pbc$, and $\triangle pca$ (see Fig. 1(a)). This operation is called *augmentation.*

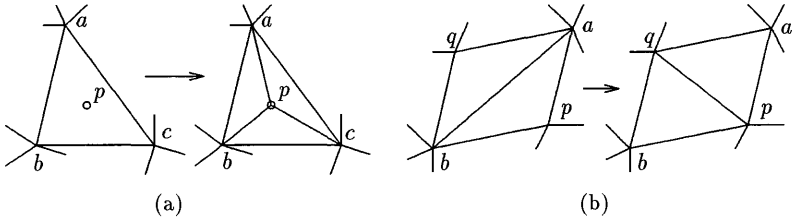

Figure 1: Incremental point insertion.

To determine whether each of these three new triangles, say $\triangle pab$, is a Delaunay triangle we perform the following Delaunay test. Let $\triangle qba$ be the triangle on the "other side" of the edge $\overline{ab}$. If either (1) no such triangle exists (because edge $\overline{ab}$ is an edge of the unit square) or (2) if the triangle does exist but p does not lie within the circumcircle of $\triangle qba$, then $\triangle pab$ is Delaunay and no further updating is needed. Otherwise replace the two triangles $\triangle pab$ and $\triangle qba$ with the two triangles $\triangle paq$ and $\triangle pqb$ (see Fig. 1(b)). This is equivalent to swapping the edges $\overline{ab}$ and $\overline{pq}$, and hence is called an *edge swap.* Continue the test, this time with the triangles $\triangle paq$ and $\triangle pqb$ until all triangles pass the Delaunay test. The Delaunay test is then performed for the for the other two triangles $\triangle pbc$ and $\triangle pca$. The correctness of this algorithm is well known (see, e.g. [5]).

Pseudocode for this algorithm is given below. Guibas, Knuth, and Sharir actually describe an elegant nonrecursive implementation of this algorithm [4]. We have presented the algorithm in this recursive form to emphasize its symmetry with the incremental deletion algorithm, which we present in the next section. The procedure invokes the primitive $\text{in}(u, v, w, p)$ which determines whether the point p lies within the circumcircle of the triangle $\triangle uvw$ (where u, v and w are given in counterclockwise order). The argument p is the point to be inserted, and D is the existing triangulation.

```
procedure Insert(p, D);
begin
    Find the triangle △abc of D containing p;
    Replace △abc by the three triangles △pab, △pbc, and △pca in D;
    SwapTest(ab, D);
    SwapTest(bc, D);
```

```
    SwapTest(ca, D);
end;

procedure SwapTest(xz, D);
begin
    if xz is an edge of the unit square then return;
    Let y be the third vertex of the triangle to the right of xz in D;
    if in(x, y, z, p) then begin
        Replace triangles △xyz, △pxz with △pyz, △pxy in D;
        SwapTest(xy, D);
        SwapTest(yz, D);
    end;
end;
```

Letting (u_x, u_y) denote the coordinate of the point u, the primitive $\text{in}(u, v, w, p)$ is implemented by evaluating the following determinant.

$$\text{in}(u, v, w, p) \quad \equiv \quad \det \begin{pmatrix} u_x & u_y & u_x^2 + u_y^2 & 1 \\ v_x & v_y & v_x^2 + v_y^2 & 1 \\ w_x & w_y & w_x^2 + w_y^2 & 1 \\ p_x & p_y & p_x^2 + p_y^2 & 1 \end{pmatrix} > 0.$$

The data structure used for storing the triangulation can be chosen from any number of standard structures for storing subdivisions of the plane, such as the quad-edge data structure [5] or the winged-edge data structure [7]. These data structures are both edge-based in the sense that the primitive objects of the data structure are the edges. In our implementation a triangle-based data structure was employed. This is particularly convenient for the triangle-based point location techniques which discussed below. The fundamental property required of any data structure for this problem is that it be able to move from one triangle of the triangulation to each of its three neighboring triangles in constant time.

One important aspect of this algorithm is the particular order in which the triangles are deleted from the triangulation. Consider the set of triangles of the original triangulation which were replaced during insertion and let $R(p)$ denote the dual graph of this set of replaced triangles (where each vertex of this graph corresponds to a deleted triangle, and two vertices are adjacent if and only if these triangles share a common edge)

LEMMA 2.1 *The dual graph $R(p)$ is a tree. Further, if we take the root to be the triangle of the original triangulation which contains p, $\triangle abc$, then the sequence of deleted triangles forms a counterclockwise preorder traversal of this tree.*

PROOF: The dual graph is a tree because the union of the set of new triangles (those having p as a vertex) defines a simple polygon. (In fact this polygon is star-shaped with respect to p.) This polygon contains no other points of the point set in its interior. Thus the set of deleted triangles forms a triangulation of this polygon. It is well known that the triangulation of a simple polygon is a tree.

The fact that the deleted triangles define to a counterclockwise preorder traversal of this tree is an immediate consequence of the facts that (1) the edge swap is performed before either recursive call is made, and (2) the two recursive calls made in

the algorithm are made in counterclockwise order relative to p. □

To analyze the complexity of the algorithm, Guibas, Knuth, and Sharir proved the following result [4]. Observe all of the triangles introduced by the insertion algorithm are adjacent to the new point p. Thus a triangle never "reappears" once it has been replaced (assuming insertions only).

THEOREM 2.1 (Guibas, Knuth, Sharir) *Let P be a set of n points in the plane, which are inserted in random order into a Delaunay triangulation using the above procedure. The expected number of triangles that appear at any time the construction is $O(n)$.*

Because the algorithm performs only a constant amount of work with each newly created triangle, it would follow that the expected running time of the algorithm is $O(n)$. However, the important missing element is the time required to determine which triangle the newly added point p lies in. It will be this point location problem which drives the total expected running time up to $O(n \log n)$. We describe two ways in which this point location can be performed.

The first method involves simple *bucketing*. Let us assume that the set of points P is known in advance. When the algorithm is initiated, the triangulation consists of a decomposition of the unit square into two triangles. We partition the initial point set into two groups, or *buckets*, depending on which triangle they lie in. As the triangulation is updated, we iteratively redistribute the points into finer and finer partitions, so that each triangle of the triangulation is associated with the set of points which lie within this triangle. (Our general position assumptions allow us to ignore the case in which a point lies on the edge of a triangle. In general this is handled by devising a rule which consistently forces all such points into one of the adjacent triangles. See also [5].) When a triangle is replaced by augmentation, only the points contained within this triangle need be rebucketed into one of three new triangles. When two triangles are replaced by two others through an edge swap, only the points in the original two triangles need be rebucketed into one of the two new triangles. (See Fig. 2(a).)

The second method was introduced by Guibas, Knuth, and Sharir. The *history* of the triangulation updates is stored. In particular, whenever a triangle $\triangle abc$ is replaced by two or more new triangles, $\triangle abc$ remains as part of the structure and marked as "old", and pointers are added from $\triangle abc$ to each of the newly generated triangles. The newly added triangles are called the *children* of the old triangles, and the old triangles are the *parents* of the new triangles. The number of children is either three (which occurs when an augmentation is performed) or two (which occurs when an edge swap is performed). Thus each node has a constant number of children.

Initially the data structure consists of a single node which implicitly represents the unit square (the only node which does not correspond to a triangle), and the insertion of the initial diagonal produces two triangular children. This process defines a rooted directed acyclic graph, which we call the *history graph*. The history graph is not a tree, because a given node may have as many as two parents in this structure (and the deletion algorithm of the next section may produce three parents).

In order to locate the triangle containing a newly added point, we start from root node representing the unit square, and trace through the chronological chain of "old" triangles containing this point until arriving at the triangle of the current triangulation which contains the point. At each "old" triangle there are at most three triangles at the next level which could contain the point, thus constant time suffices to determine the next triangle of the chain in which the point lies. (See Fig. 2(b).)

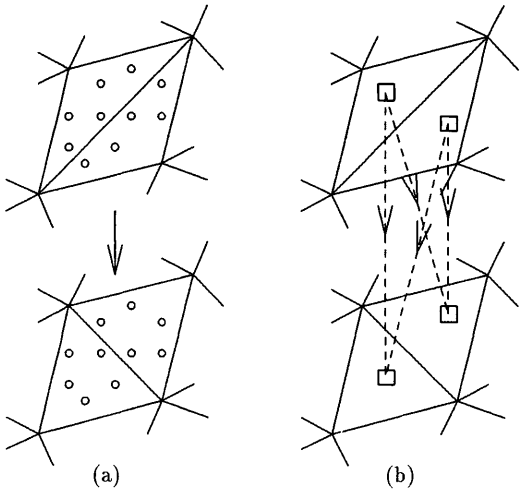

Figure 2: Point location.

Under the assumption that all points of the base set are inserted into the triangulation, the total time required by the bucketing and the history methods are identical, since the same discriminating tests are made for each point, and each point moves through the same sequence of triangles in each method. This history approach can be viewed as a sort of *lazy evaluation* of the bucketing scheme since it is only applied to the points which are indeed added to the triangulation. Thus if not all of the points are added to the final triangulation, the history method has an advantage over bucketing with respect to execution time. In addition this method need not know all the points in advance. The number of times a point is moved from one triangle to another can be as large as $O(n)$ per insertion. However, Guibas, Knuth, Sharir show that the number of triangles through which a point moves, when averaged over all the points and all insertions, is only $O(\log n)$ in the expected case. From this it follows that the incremental algorithm runs in $O(n \log n)$ time in the expected case, irrespective of whether the bucketing or history method is used.

One disadvantage of the history method is that its space usage is dependent on the number of edge swaps performed by the algorithm. Although this number is $O(n)$ in the expected case, it could be as large as $O(n^2)$ (although the probability of this occurring for large n is extremely small under the assumption of random insertion.) The bucketing method has the advantage that it never requires more than $O(n)$ space in the worst case even if the point insertion violates the randomness assumption. This is true because only the current triangulation is stored.

One big advantage of the history method is a type of *persistence*. Persistence refers to the ability of a data structure to maintain its history. In this case, by storing history of the data structure it is an easy matter to restore a recent version of the data structure. This is done quite simply by reversing the sequence of edge swaps by walking backwards through the history graph. Since the number of edge swaps per insertion is expected to be a constant (and we will see that the same holds true for deletion), the time needed to restore an earlier version of the triangulation

is proportional to the number updates performed between the earlier version and the present one. This is a $\log n$ factor savings in running time over the naive method of reversing the string of recent operations. This same persistence will apply for deletions also as we shall see in the next section.

3 Incremental Deletion

In this section we introduce a simple incremental algorithm for deleting a point from a Delaunay triangulation. It seems inherently harder to implement a purely incremental deletion algorithm in the spirit of the insertion algorithm given in the previous section. Our deletion algorithm applies the insertion algorithm of the previous section in an off-line mode to compute an intermediate Delaunay triangulation, which it then uses to guide an incremental sequence of edge swaps to perform the actual deletion. Our algorithm has the interesting property that, with careful implementation (and assuming that points are in general position), it swaps edges in essentially the reverse order from the insertion algorithm. Thus, by calling the deletion algorithm on the points in the reverse order of insertion, the algorithm will incrementally disassemble the triangulation in exactly the reverse order of its assembly.

As before, let $P = \{p_1, p_2, \ldots, p_n\}$ denote a set of points in the plane (including the vertices of the unit square) and let $D(P)$ denote the Delaunay triangulation of this point set. Let $p \in P$ be the point to be deleted. We assume that p is not one of the vertices of the unit square. We make the same general position assumptions of the previous section that no three points are colinear and no four points are cocircular.

Let T denote the set of triangles incident to p in the Delaunay triangulation. Because p is not a vertex of the unit square, p does not lie on the convex hull of P, and hence the union of the triangles of T is a star-shaped polygon containing the point p in its interior (and in fact within its kernel). Let Γ denote this polygon. Observe that any triangle in T cannot be part of the triangulation after the deletion of p, and that any triangle in $D(P) - T$ (i.e. any triangle which is not incident to p) is still empty after the deletion of p. Thus, only the region of the plane covered by the polygon Γ need be retriangulated.

We begin by outlining a nonincremental algorithm, which we will shortly modify to give an incremental algorithm. By a cyclic enumeration of the triangles of T, determine the boundary vertices of the star-shaped polygon Γ. Compute the Delaunay triangulation D_Γ of the polygon Γ by any algorithm (see the remark below). Replace the triangles of T by the triangles of D_Γ giving the new Delaunay triangulation $D(P - \{p\})$.

Unfortunately, this algorithm is not incremental, and it is unclear how to modify the point location algorithms to deal with the sudden replacement of potentially $O(n)$ triangles by $O(n)$ new triangles. However, imagine that p were the last point of the triangulation to be inserted, prior to this deletion. The insertion of p would induce a particular sequence of edge swaps mapping $D(P - \{p\})$ to $D(P)$. Since we know both triangulations, it is a relatively simple matter to perform the edge swaps in reverse order to transform $D(P)$ incrementally to $D(P - \{p\})$.

We solve this problem by a simple leaf pruning method. Recalling the discussion preceding Lemma 2.1, a triangle of D_Γ is a *leaf* of the dual graph of D_Γ if and only if at least two of its sides lie on the boundary of Γ. From this lemma we know that by the preordering of replaced triangles, the edge swaps are performed in such an order that the leaves of Γ are the *last* triangles to be replaced in the triangulation. Thus,

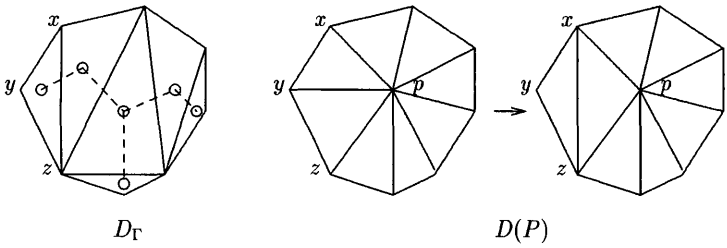

Figure 3: Leaf Pruning.

to "undo" the effects of the insertion algorithm we locate a leaf triangle $\triangle xyz$ of the D_Γ and remove this triangle *first*. Let us assume that the vertices of $\triangle xyz$ are given so that x and z are neighbors of y along the boundary of Γ). Assuming that $\triangle xyz$ does not contain p, swap the edges $\overline{xz}$ and $\overline{py}$ in the triangulation T (see Fig. 3). As a consequence, the vertex y is no longer adjacent to p. We can eliminate y from Γ, by connecting x to z, and apply the algorithm iteratively to the remaining polygon.

The complete deletion algorithm is given below. The argument p is the point to be deleted, and D is the existing triangulation. Leaf pruning is performed recursively, to emphasize its symmetry with the insertion algorithm. (Although, as in Guibas, Knuth, and Sharir [4], there does exist a purely iterative solution.)

procedure Delete(p, D);
begin
  Find the set of triangles $T \subseteq D$ incident to p;
  Let Γ be the polyon defined by the union of T;
  Compute D_Γ, the Delaunay triangulation of Γ;
  Let $\triangle abc$ be the triangle of D_Γ which contains p;
  UnSwap($\overline{ca}, D_\Gamma, D$);
  UnSwap($\overline{bc}, D_\Gamma, D$);
  UnSwap($\overline{ab}, D_\Gamma, D$);
  Replace the three triangles $\triangle pab$, $\triangle pbc$, and $\triangle pca$ by $\triangle abc$ in D;
  Delete the triangulation D_Γ;
end;

procedure UnSwap($\overline{xz}, D_\Gamma, D$);
begin
  if $\overline{xz}$ is an exterior edge of D_Γ **then return**;
  Let y be the third vertex of the triangle to the right of $\overline{xz}$;
  UnSwap($\overline{yz}, D_\Gamma, D$);
  UnSwap($\overline{xy}, D_\Gamma, D$);
  Replace triangles $\triangle pyz$, $\triangle pxy$ with $\triangle xyz$, $\triangle pxz$ in D;
end;

REMARK: The most efficient way to construct the Delaunay triangulation of Γ theoretically is by the rather sophisticated linear time algorithm by Aggarwal, Guibas, Shaxe, and Shor [1]. (Although this algorithm is designed for computing the Delaunay

triangulation of a convex polygon, it is shown in [1] that it can be applied to patch up a Delaunay triangulation when a point is deleted.) A much simpler but theoretically less efficient way to compute this Delaunay triangulation is to apply the incremental insertion algorithm of the previous section to the vertices of Γ given in random order. It is quite easy to show that the boundary of Γ remains intact in this triangulation (because each edge of Γ was Delaunay prior to the deletion of p), thus the triangulation of the interior of Γ can be determined by discarding all triangles which lie outside of the polygon.

This expected case $O(n \log n)$ algorithm is theoretically slower than the linear time algorithm. However, since the expected degree of a vertex in a planar graph is less than six (by Euler's formula), practically speaking the minute loss of asymptotic running time is more than compensated for by the lower constant of proportionality of the simple incremental algorithm together with the significant savings of programming effort.

Our practical experience has shown that for many natural point distributions, the maximum degree in a Delaunay triangulation rarely exceeds 16, independent of n. Thus, it may not be entirely unreasonable to apply an $O(n^2)$ triangulation algorithm. We decided against this approach because (1) the incremental algorithm is already available for our use, and (2) if there is even one vertex of degree $\Omega(n)$ in the triangulation, then the expected running time of the deletion algorithm would grow to $O(n)$. This is considerably worse than the $O(\log n)$ expected case bound, which we show below.

Given the structure of the deletion algorithm, it is relatively easy to see that it creates triangles in exactly the reverse order of the insertion algorithm, namely in a clockwise postorder traversal of the dual tree of D_Γ. If the points are in general position, and the choice of the orientation of the triangle $\triangle abc$ in procedure Delete is chosen to be identical to the triangle $\triangle abc$ of procedure Insert, then the deletion algorithm effectively swaps edges in the reverse order as the insertion algorithm (assuming that the point deleted is the last point inserted). If the points are not in general position (in particular, if four or more points are cocircular) then there may be multiple final Delaunay triangulations for Γ, thus we cannot guarantee that sequence of edge swaps is the same. We can force the orientation of the triangle $\triangle abc$ to be identical in both cases by selecting the point a in some canonical manner (e.g. by taking the point whose coordinates are lexicographically maximal).

Observe that the deletion algorithm does not need to deal with point location (except perhaps at the level of the user-interface in order to determine which point is to be deleted). However, if subsequent insertions are to be performed, the point location structures described in the previous section must be updated to reflect the change in the triangulation.

When each edge swap is performed we handle it in exactly the same way that we handled an edge swap in the case of insertion. For point bucketing the points contained within the affected triangles are redistributed among the new triangles. For the history graph the affected triangles are marked as "old" and pointers are added to the new overlapping triangles. The final step of the deletion algorithm, in which the triangles $\triangle pab$, $\triangle pbc$, and $\triangle pca$ are replaced by $\triangle abc$, is the inverse of the augmentation step seen in the insertion algorithm. For the bucketing method, the three sets of buckets are merged into a common bucket for $\triangle abc$. For the history graph method, we store a single pointer from each of the three old triangles to the newly created containing triangle.

Our next task is to analyze the complexity of the deletion algorithm. This task is complicated by the fact that the analysis of the insertion algorithm was based on the assumption that only insertions are performed and that all points are eventually added to the triangulation. To appreciate the difficulties arising when insertions and deletions can be combined, consider the case in which a single point is inserted into the triangulation and then it is deleted. This process is repeated a large number of times, N. If the bucketing method of point location is used, then when a single point is inserted all the points in the base set must be rebucketed, requiring $O(n)$ time. If the history method is used then the history graph degenerates into a structure of depth $O(N)$. Thus the expected running time in either case is much worse than the desired $O(\log n)$.

In the next section we show how to deal with the question of point location in such dynamic situations. For now, we analyze the expected running time of one deletion. This running time follows almost directly from the analysis of the insertion algorithm. Because the particular edge swaps and point movements (arising from point location) for deletion are just the reverse of those for insertion, the expected number of edge swaps and point movements needed to delete a random point p from a triangulation $D(P)$, is identical to the expected number of edge swaps and point movements needed to insert the random point p into the triangulation $D(P - \{p\})$. Under our assumptions of random point insertion and deletion, the sets P and $P-\{p\}$ are random point sets. Thus, this portion of the cost of deleting a random point from a triangulation n points is $O(\log n)$ in the expected case.

The only other aspect of the complexity of deleting a point p is the cost of computing the Delaunay triangulation of Γ, the polygon of neighboring vertices. The number of vertices in Γ is equal to the degree of p in the triangulation. To establish the expected cost of this operation, let $d_1, d_2, \ldots, d_n$ denote the degrees of each of the n vertices of the triangulation. By Euler's formula we know that the sum of degrees, which equals twice the number of edges in the graph, is at most $6n$. By the analysis of the previous section, the expected time to compute the Delaunay triangulation of a set of d_i points is $O(d_i \log d_i)$. Thus, the expected time needed to compute one such Delaunay triangulation, under the assumption that each point is equally likely to be deleted is

$$\frac{1}{n}\sum_{i=1}^{n}(d_i \log d_i) \leq \frac{1}{n}\left(\sum_{i=1}^{n} d_i\right)\log n \leq \frac{6n}{n}\log n = 6\log n.$$

Thus, the expected time to delete a point from the triangulation is $O(\log n)$, from which we have our main result.

THEOREM 3.1 *Given the above deletion algorithm (ignoring point location issues) a point can be deleted from a Delaunay triangulation of n points in expected $O(\log n)$ time, under the assumption that each point of the triangulation is equally likely to be deleted.*

4 Sequences of Insertions and Deletions

As we mentioned in the previous section, in the worst case, where long sequences of insertions and deletions are made, the expected case running time of the algorithm can be much larger than $O(\log n)$. In this section we consider how to deal with the problem.

Our first observation is that in certain relatively benign cases (which may be quite common in many practical applications) there is really no problem at all. For example, if insertions are more common than deletions (in the sense that the ratio of the number of insertions to deletions is strictly greater than unity) then it follows that over a long sequence, the cost of updating the triangulation is dominated by the costs of the insertions. Although the deletions cause an increase in the size of the history graph, the assumption of randomness implies that these variations in the history graph are distributed evenly throughout the graph, and it was shown in the previous section that the local effect of each deletion on the structure is essentially equivalent to the effect of an insertion.

In steady state situations, where the number of active points in the triangulation reaches an equilibrium, a direct application of the deletion and insertion algorithms is rather unpredictable. If the number of active points is a roughly a constant fraction of the total number of base points, then the randomness of insertion and deletion, combined with Guibas, Knuth, and Sharir's arguments about the widths of triangles, imply that only a constant number of points will be rebucketed with each change to the triangulation. However, if the number of active points is significantly less than the total number of base points, then nearly all of the nonactive base points may be rebucketed with each update. The history method will fair even worse, because irrespective of the number of active points, the history graph grows without bound as updates are made.

In this section we will consider how to periodically rebalance the history graph so that these problems can be avoided. The idea is that from time to time, we will completely reconstruct the history graph from scratch for only the current set of active points, and destroy the old graph. (Observe that this will have the unfortunate consequence of destroying the persistence property provided by the unpruned structure.) We refer to this process as *reorganization*. Let n denote the number of active (triangulation) points. We show that by applying reorganization at appropriate times, we can maintain an $O(\log n)$ expected time cost for insertion or deletion, when amortized over sequences of insertions and deletions. The expectation here is over possible random choices of which point to insert or delete. The choice of whether to insert or to delete is arbitrary.

We assume initially that the triangulation is trivial (consisting only of the four vertices and two triangles of the unit square). Let t denote the total number of insertion/deletion requests which have been performed, and let $n(t)$ denote the number of active points in the triangulation after the t-th request. Thus, $n(0) = 4$. Let t_0 be the time (i.e. the request number) of the last reorganization. After performing the t-th operation we test whether

$$t - t_0 > n(t).$$

If this is the case then reorganization is performed. Reorganization consists of first discarding the existing history graph and triangulation, and then constructing a new history graph and Delaunay triangulation by inserting (in random order) each of the current active points.

THEOREM 4.1 *Using this reorganization scheme, the expected amortized time for processing an insertion or deletion request for a random point is $O(\log n)$, where n is the number of active points at the time of the insertion or deletion.*

PROOF: The theorem follows from two observations. The first is that if periodic reorganization is applied, then the execution time of insertion or deletion at any time

(ignoring reorganization) is $O(\log n)$. The second observation is that reorganization is performed infrequently enough that it does not increase the asymptotic running time of the algorithm.

To prove the first observation, let n_0 denote the number of active points at the time of the most recent reorganization. Because this is the first point at which we have performed a reorganization since t_0, it follows that for all s, $t_0 \leq s < t$, $n(s) \geq s - t_0$.

The number of nodes in the history graph increases by an expected constant amount with each insertion or deletion (since the expected number of new edge swaps is constant). Thus the expected size of the history graph after the s-th request is roughly proportional to $n_0 + (s - t_0)$.

Since we can lose at most one point at each insertion/deletion request, we have $n(s) + (s - t_0) \geq n_0$. Thus

$$n_0 + (s - t_0) \leq n(s) + 2(s - t_0) \leq n(s) + 2n(s) = 3n(s).$$

In other words, at no time is the expected size of the history graph significantly larger than the number of active points. Because changes to the history graph are made randomly throughout its structure, it follows that the cost of searching the graph does not increase asymptotically, so the search time is $O(\log(n_0 + s - t_0)) = O(\log n(s))$. All other aspects of the insertion and deletion routines are $O(\log n(s))$ running time. This establishes the first observation.

To establish the second observation, observe that the expected time to perform reorganization is $O(n \log n)$, where n is the number of active points. Since the expected case of insertion or deletion ignoring reorganization is $O(\log n)$, it follows that the cost of reorganization will not dominate the overall cost if the number of insertions or deletions since the last reorganization is at least as large as $n = n(t)$. However, in order to perform reorganization it must be the $t - t_0 > n$, thus the number of requests which have been processed since the last reorganization is at least as large as the number of active points. This establishes the second observation. □

5 Implementation Experience

In this section we discuss our implementation of the algorithm. The algorithm has been implemented in the C programming language, under the Unix operating system. (Currently the reorganization scheme has not been implemented.) It has been designed to provide statistics on the execution of the algorithm for the purposes of evaluating its efficiency. Rather than measuring execution time by CPU seconds, because of its dependence on the particular machine and compiler, we have measured two quantities which we feel give a strong indication of the algorithm's general performance. First, we have measured the number of times that a point moves from one triangle to another in the bucketing algorithm (equivalently, the number of levels that each point travels through the history graph), and second we have measured the number of edge swaps which were performed.

We have run the following experiments involving point insertion. (The reasons that we did not consider deletion are (1) the number of edge swaps and point movements for deletion are identical in the expected case to insertion, and (2) we have not yet implemented the reorganization scheme described in the previous section.)

Uniform Data: Points were sampled from a uniform distribution over the unit square and inserted into the triangulation. Point sets of size 50, 100, 200, 400, 800, 1600, 3200, and 6400 were considered.

Gaussian Data: Points were sampled from a Gaussian distribution whose center is at the center of the unit square and whose standard deviation was 0.2 in each of the x and y directions. Point sets of the same sizes as in the uniform case were considered.

Sorted Data: To test the sensitivity of the algorithm to violations of the randomness assumption, we ran an experiment in which points were selected uniformly from the unit square, but were inserted in order of increasing x-coordinate.

Partially Random Data: In this variant of the previous experiment, we inserted points in which the first p points were inserted randomly (out of a total of 6400), and the remaining points were inserted in order of increasing x-coordinate. The values of p tested were 25, 50, 100, 200, 400, 800, 1600, 3600, and 6400.

The results of the these experiments are given below.

Uniform Data: Fig. 4(a) shows a plot of the number of edge swaps performed by the algorithm versus n, the number of points. The regression line fitted to the data is $2.97n - 68.0$. Fig. 4(b) shows a plot of $\log_{10} n$ versus the average number of point movements per point. Standard deviations are indicated by vertical lines. The regression line fitted to the data is $9.02 \log_{10} n - 3.75$.

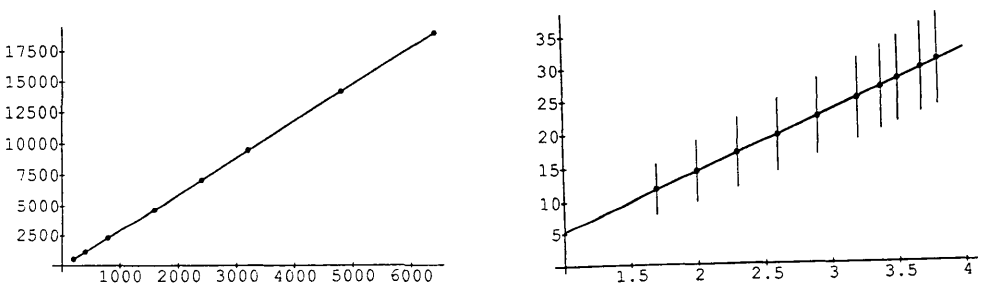

Figure 4: Uniform data: Total edge swaps and average point moves.

Gaussian Data: Analogous results for Gaussian data are shown in Figs. 5(a) and (b). In the first case the regression equation is $3.01n - 77.4$, and in the second case it is $9.66 \log_{10} n - 6.32$. Both cases are in close agreement with the uniform case, although the number of edge swaps is slightly larger in the Gaussian case.

Sorted Data: Fig. 6(a) shows a plot of the total number of edge swaps performed versus n in the case that points are inserted in sorted order. The regression line fitted to the data is $4.72n - 494$. The slope is greater than the previous cases, however this supports the observation made by Tipper [9] that the average number of edge swaps per point is independent of n. However, the plot of $\log_{10} n$ versus the number of point moves showed a striking nonlinear behavior (see Fig. 6(b)). It is clear that the assumption of randomness is critical to the analysis of the point location schemes.

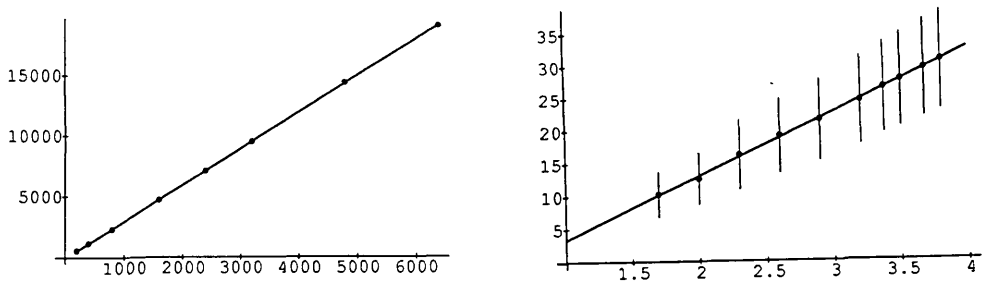

Figure 5: Gaussian data: Total edge swaps and average point moves.

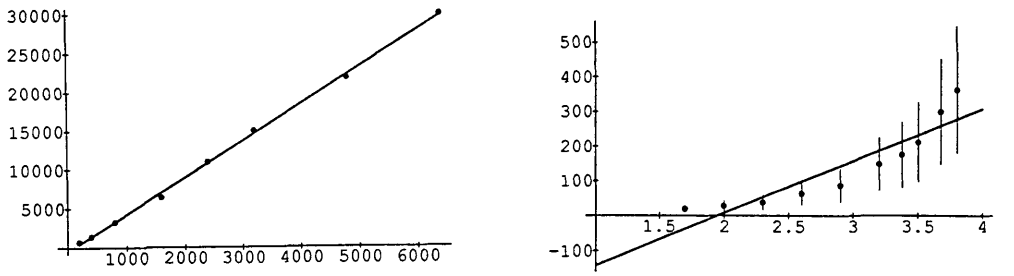

Figure 6: Sorted data: Total edge swaps and average point moves.

Partially Random Data: The results of the previous experiment lead us to the question of how many initial points need be inserted randomly in order to guarantee fairly good performance in point location. Fig. 7 shows $\log_{10} p$ versus the average number of movements per point. Interestingly, with as few as 200 of the 6400 points inserted randomly (about .03% of the total) the performance is within a factor of 2 of the totally random case.

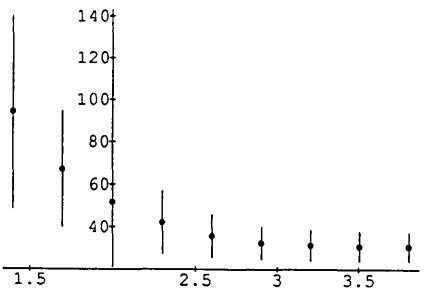

Figure 7: Partially random data: Average point moves.

6 Conclusions

We have presented and analyzed the complexity of a procedure for computing and updating Delaunay triangulations for point insertion and deletion. The algorithm

is randomized and incremental, based on a recent algorithm of Guibas, Knuth, and Sharir. The algorithm has the nice feature that it is asymptotically as efficient (in the expected case) and yet much simpler than standard divide-and-conquer algorithms. Its expected running time is independent of the distribution of the points, only on the order in which the points are inserted or deleted. We have implemented a portion of the algorithm for the purpose of empirical analysis. Our studies seem to indicate that the running time is quite good even if the assumption of random insertion order is violated as long as an initial fraction of the points are inserted randomly.

One interesting open problem raised by this research is whether these results can be applied to more general types of triangulations. In particular, in geographic information systems, it is quite common to require that certain edges be present in the Delaunay triangulation, giving rise to a *constrained Delaunay triangulation.* It would be of interest to develop and analyze the performance of a randomized incremental algorithm for constrained triangulations.

References

1. A. Aggarwal, L. J. Guibas, J. Saxe, P. W. Shor, A linear-time algorithm for computing the Voronoi diagram of a convex polygon, *Discrete and Computational Geometry* 4 (1989), 591–604.

2. S. Fortune, A sweepline algorithm for Voronoi diagrams, *Algorithmica*, 2 (1987), 153–174.

3. P. Green and R. Sibson, Computing Dirichlet tesselation in the plane, *Comput. J.* 21 (1977), 168–173.

4. L. Guibas, D. Knuth, and M. Sharir, Randomized incremental construction of Delaunay and Voronoi diagrams, Unpublished manuscript (1990), also appeared in the *Proceedings of ICALP*, 1990.

5. L. Guibas and J. Stolfi, Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams, *ACM Trans. on Graphics*, 4 (1985), 74–123.

6. M. Heller, Triangulation algorithms for adaptive terrain modelling," *4th Symposium on Spatial Data Handling*, 1990, 163–174.

7. M. Mäntyllä, *An Introduction to Solid Modeling*, Computer Science Press, Rockville, Maryland, 1988.

8. T. K. Peucker, R. J. Fowler, J. J. Little, and D. D. Mark, Digital representation of three dimensional surfaces by triangulated irregular networks. Tech. Report #10, ONR Contract N00014-75-C-0886, 1976.

9. J. C. Tipper, A straightforward iterative algorithm for the planar Voronoi diagram, *Information Proc. Letters* 34 (1990), 155–160.